Technical Report
COMP1100 Assignment 3

Jacob Bos
ANU u7469354

May 29, 2022

Lab: Tuesday 11am

Tutor: Abhaas Goyal

Word-count beyond cover page at $\leq$ 1500 words

Contents

1 Introduction

The program detailed herein is an implementation of a few AI's for solving Fanorona with complimentary unit tests.

2 Documentation

Design Documentation and Technical Decisions

First capture move , used to test the greedy AI, is an AI that takes the head a list of possible capturing moves provided by `captures` else returning the first legal move.

The Greedy AI is conceptually simple. The main function `greedy` cases on the turn in the gamestate and picks the move that either maximizes (for Player1) or minimizes (Player2) the heuristic value. It does this using `greedyHelp` that recurses through a list of move/value pairs using an accumulator and the appropriate evaluator to pick the move. This move/value list is generated by a mapping of `diffPieces` and `applyMove` to the `legalMoves` list.

The first Minimax uses two trees, `GameTree` stores all the possible gamestate evolutions and is generated through an infinite recursion in `gameTree` which takes a state, puts it into a node and then maps `gameTree` to all its child states generated through a mapping of `applyMove` to a list of `legalMoves` which has `Nothing`s recursively filtered out by `purge`.

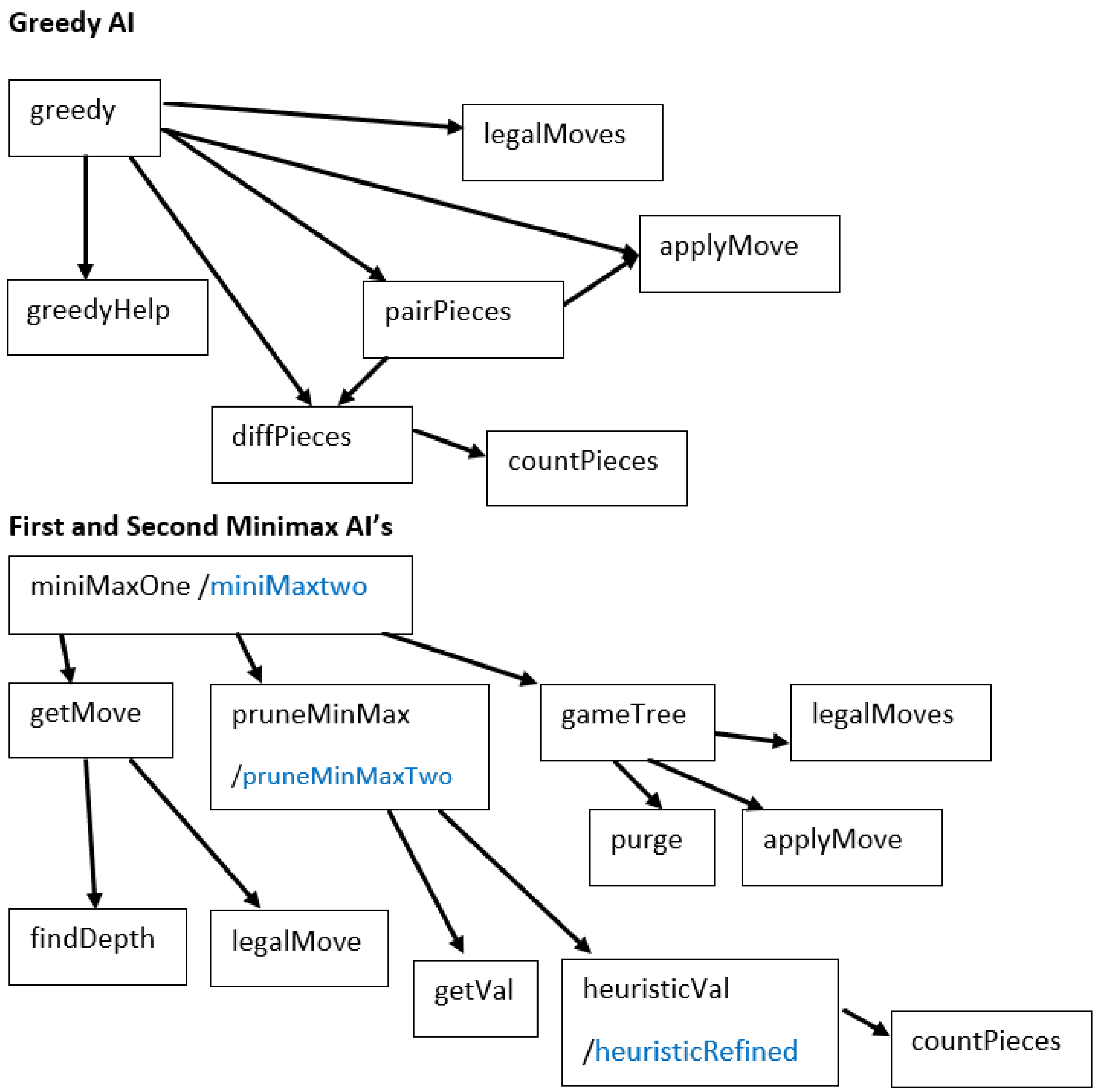
`EvalTree` is similar to `GameTree` but stores a value corresponding to the best possible heuristic value for the player who's turn it is at a node, and is pruned to a given *move depth.* The `EvalTree` is generated by `pruneMinMax` recursively navigating `GameTree`, casing on the integer depth given to the function, if the depth is zero or the game is over then it evaluates the heuristic value at that node and then terminates that branch. Else, it cases on the state held in the node. If the state at a node contains `Turn Player1` then it assigns the maximum of the values in its child `EvalTree` nodes else if the turn is the `Player2` it assigns the minimum value of its child nodes to the given node. This results in the best possible outcome for the player in the initial state ending up in the head node. The heuristic value, calculated by `heuristicVal`, is the difference in the number of pieces between players given by `countPieces`.

To retrieve the best move we note that the best move is at the same depth in the `legalMoves` list as the value stored in the head node is in the list of child nodes. This is because the list of child nodes is produced by a mapping on the `legalMoves` list. Consequently to find the best move, we extract the value from the head of the output of `pruneMinMax` using `getVal` and then find its depth recursively in the list provided by mapping `getVal` to the child nodes using `findDepth`. Consequently `getMove` uses (!!) to extract the best move at its expected depth in the `legalMoves` list.

The second Minimax Is identical to the first except that the heuristic function assigns win or loss gamestates with values more extreme than typical. Doing so improved performance by a little, weighting winning more heavily.

Program Design

The AI structures are graphed below:



The Greedy AI currently does a lot of unnecessary work with Maybe types which could be eliminated with the strategies used in the minimax AIs that were later developed. The maybe types used require extra case statements and so it was necessary to break the AI into a number of helpers to improve readability.

Both MiniMax AIs share the same structure and most of their functions. Both are called by a top end function that inputs the initial state and calls a function that gets the move based on the result of the pruner function's evaluation tree. It was chosen to move `getVal` to a helper function to reduce the population of things in the pruner's `where` clause.

Assumptions

The primary assumption made is that the ordering of the `legalMoves` list is the same as the ordering of their values in the child nodes of `EvalTree`. This assumption is reasonable because there should be no `Nothing`s in the list for `purge` to remove because they result from an illegal move being fed to `applyMove` which is impossible using `legalMoves`. Thus the values in the first child nodes are just repeated mappings onto the `legalMoves` list, preserving order.

3 Testing

Unit tests aimed to cover as many cases on as many functions as possible. Unfortunately, many of the functions deal with complex datatypes made writing typical arguments difficult. Consequently some tests use `initialState` and compare the lengths of arguments and outputs.

The Greedy AI has two test groups. `pairPieces` was tested by checking that the list of pairs of legal moves and their associated heuristic value is the same length as the list of `legalMoves`. The passing of this test asserts the function's correctness. `diffPieces` is tested against two cases, that it returns `Nothing` for argument `Nothing` and `Just 0` for an argument of just `initialState`.

The MiniMax AI has four test groups. Firstly `purge` has four test cases. They were made easier to write because `purge` is polymorphic and so a simpler input type of `Int` was used. `purge` is then tested against a case of the empty list, all `Nothing`s, all `Just x`'s and a mix of `Nothing`s and `Just`s. Passing these tests implied that the function was correct. Similarly, since `findDepth` is polymorphic it was easier to write test cases. It was not tested against the empty list since that returns an error but was tested against cases where the element occurred once or twice in the list where its supposed to take the first. Thirdly, `getVal` was tested against one test case, indicating its ability to correctly retrieve a nodes's value. Lastly `heuristicVal` was tested with the initial state as an argument to ensure that the output is zero as would be expected. Unfortunately it was difficult to test it against any other inputs as other arguments are difficult to write.

Performance tests were done by playing my AIs against themselves and the tournament's course AIs.

The correctness of the Greedy AI was confirmed by it beating both `firstlegalMove` and `firstCaptureMove` as it should statistically behave better than both. Whilst a faulty Greedy may be able to beat FLM by chance it is less likely to beat an FCM by chance especially if it was accidentally minimizing when it was supposed to maximize or vices-versa. Since the Greedy was also able to consistently beat me as a human I considered that it was likely correct.

The MiniMax AI's were tested against both the greedy AI and the Course AIs, `miniMaxtwo` consistently outperformed `greedy` and all of the course greedy AIs except for third where playing first results in a draw. `MiniMaxTwo` majority draws against the course Minimaxes and Alpha-beta pruners further indicating correctness. It would be expected with such a simple heuristic that another minimax or minimax-alpha/beta AI with a better heuristic would slightly outperform.

4 Reflection

Design Choices

For the `greedy` and `miniMaxOne` it was decided to use the difference of pieces as the heuristic due to its easy implementation using `countPieces` from `Fanorona.hs`. This was done to allow greater time to design and understand the algorithms. The second Minimax used the updates described in an attempt to prioritize winning moves in endgame. It was chosen that the greedy AI should pair moves with their associated value to allow for an accumulator recursion to find the move with the best outcome. In contrast the minimax AI's did not store moves in the structure, relying on the discussed assumption about list lengths. This was done in order to simplify the structures and functions as much as possible in an attempt to improve style and speed, being able to use pre-optimized functions like `maximize` and avoid convoluted datatypes. The minimax pruners were designed to make as good use of laziness as possible. This allowed a node to be assigned the minimum of maximum value of its children before the children were evaluated. The structure of each AI was dictated by the authors thought process and the ideas that came to them. Functions were refined in the ways that the author felt were suitable and enhanced style.

Reflection

Upon reflection the author would have designed the pruner to take a heuristic function as an argument so as to not need to rewrite the pruner function to implement a different heuristic. Further, they would have removed the states from the nodes of the `EvalTree` to simplify the structure. It would be beneficial to implement a pruning strategy such as alpha-beta to increase the possible search depth. Also, time would be spent developing a heuristic based on a deeper understanding of the game to evaluate the worth of different arrangements of pieces. Unfortunately they had no time to invest in developing an alpha-beta beta pruner and struggled to decide how to modify the current pruning function, undecided as to whether keep track of the values in the tree or accumulated in the pruner function's arguments.